A Portable Real-Time Kernel in C

Just how much is a real-time multitasking kernel worth to you? If you're like many developers, the benefits of using a commercial real-time operaing system or executive are obvious. In exchange for a licensing fee, you get a debugged product with documentation and technical support. Unfortunately, many developers find themselves in a SituatiOn where they need a real-time operating system, but have virtually no budget for purchasing one. What do you do in that situation? For many developers, the answer lies in creating their own kernel.

Rolling your own executive or operating system can be appealing. Everyone likes to create something new. But this approach isn't always practical and, in some cases, can be downright dangerous. For example, it isn't often that you're given enough time to create an operating system and complete your original project. On the few occasions when your schedule is changed to reflect the extra work, the schedule may still be overly optimistic. (It is a sad but well-established rule in our business that programmers will always underestimate the time a project will take.) Worse, when you roll your own executive, you're also volunteering for extra debugging and maintenance.

Despite the dangers I just mentioned, developers continue to take the plunge and develop their own operating systems. I know this for a fact, because I've done it myself! The good news is that you don't have to repeat the exercise. In this article, I'm going to explain the design of uCOS, a portable, preemptive multitasking kernel for microcontrollers. I've written this kernel almost entirely in C, and its performance is comparable to many commercial kernels. Using my source code, you'll be able to avoid some of the costs of creating your own kernel and spend more time on your application.

The creation and documentation for even a modest kernel can become an overwhelming task. To keep things simple, this month, I'll be discussing the kernel internals and its multitasking model at a relatively high level. Next month, I'll continue with a look at the various kernel services.

THE NATURE OF THE PROJECT

UCOS stands for "microcontroller operating system." The initial goals for uCOS were to create a small yet powerful kernel for Motorola's 68HC11 microcontroller. Specifically, I wanted a ROMable real-time kernel with preemptive multitasking. As the design evolved, however, I removed some of the target-specific limitations of uCOS to make it portable to other microprocessors. Using this approach, I was able to develop and test uCOS on a PC using Borland International's Turbo C + +. For this reason, the implementation of uCOS shown in this article is for an Intel 80186/88 microprocessor (using the small memory model).

The Intel 80186 was chosen as a target over the 8086 because the latter is a poor choice for embedded control applications. An 8086/88 design would require additional support circuitry. A minimal 80188 system can be built with four chips (80188, RAM, EPROM and 8255 PPI).

UCOS FEATURES

Although it was developed on an IBM-PC compatible, uCOS is targeted for embedded systems that use microcontrollers. The kernel is written almost entirely in C, with microprocessor-specific code written in assembly language. A small amount of assembly language was necessary but was kept to a minimum.

The uCOS source code is separated into five files. Three of these files contain target-specific code. In our 80186 example, UCOS186.C contains the task-creation function, UCOS 186.H is a header file, and UCOS186A.ASM contains the system-startup and context-switching code. (The files mentioned here are in the UCOS.ZIP archive file on the Embedded SystemsProgramming BBS at (415) 905-2389 or in library 12 of CLMFORUM on CompuServe.) These files are where the target specifics are located.

The uCOS source code that is not target-specific is in the two remaining files, UCOS.H and UCOS.C. The header file contains the system constants, data types, and function prototypes, and UCOS.C file contains the system variables and functions.

Because of its organization, uCOS can be ported to just about any microprocessor. The only requirements are that the microprocessor provides a stack pointer and the CPU registers can be pushed onto and popped from the stack. Also, your C compiler must be ANSI compatible and provide in-line assembly language or extensions that allow you to enable and disable interrupts.

uCOS is priority driven and always runs the highest priority task that is ready to run. uCOS is also fully preemptive, which means that interrupts can suspend the execution of a task and, if a higher priority task is awakened as a result of the interrupt, it will run as soon as the interrupt completes. uCOS also allows nested interrupts. Despite its flexibility, I find uCOS comparable in performance to commercially available packages. For a 10MHz 80188 system, interrupt latency is less than 450 CPU bus cycles.

Another important characteristic of a real-time kernel is determinism. The execution time of uCOS functions does not depend on the number of tasks in an application; the task-scheduling time is constant. uCOS supports systems with up to 63 user-definable tasks. Since uCOS was originally targeted for an 8-bit microcontroller, only required kernel services can be included in an application, which keeps the amount of EPROM and RAM to a minimum. The stack size for each task can also be independently specified, which further reduces the amount of RAM required.

uCOS provides a number of system services, including mailboxes, queues, semaphores, and time-related functions. Most of these system services will be covered in next month's article. The kernel is also very easy to set up, as will be demonstrated later.

I'll be the first to admit that uCOS has certain limitations. For example, tasks that execute under the kernel must have a unique priority number. No two tasks can have the same priority. Also, uCOS system calls cannot be made from nonmaskable interrupts because the kernel needs to disable interrupts to execute critical sections of code. (By the way, this limitation also exists in most commercially available kernels.) The execution time of the function used to process a system tick depends on the number of tasks in an application.

CONVENTIONS

The following conventions are adopted in uCOS. All variables, functions, and macros in uCOS start with 0S (for example, 0S-Start(), OSInit(), OS_STAT_RDY, and so on). This way, uCOS functions can be easily identified in the user's application code.

Since the kernel is written in C, the size of certain data types are target-compiler specific. To avoid confusion about the size of certain quantities, the following data types are defined for the 80186/188 in UCOS186C.H:

* BYTE: signed 8-bit value

* UBYTE: unsigned 8-bit value

* WORD: signed 16-bit value

* UWORD: unsigned 16-bit value

* LONG: signed 32-bit value

* ULONG: unsigned 32-bit value

TASK CONTROL BLOCKS

Tasks under uCOS are characterized by the task control-block data structure OS_EB, which is found in UCOS.H. An OS_TCB is used to hold the state of a task and other system-related information.

OSTCBStkPtr contains a pointer to the current top of the stack for the task. uCOS allows each task to have its own stack but, just as important, each stack can be of any size. Some commercially available kernels assume that all stacks are the same size unless you write complex hooks. This limitation becomes wasteful of RAM when all tasks have different stack requirements because the largest anticipated stack size has to be allocated for all tasks.

OSTCBStat contains the state of the task. When OSTCBStat is zero, the task is ready to run. Each bit within OSTCBStat has a special meaning (see OS_STAT...).

OSTCBPrio contains the task priority. A high-priority task has a low priority number (the lower the number, the higher the actual priority).

OSTCBDly is used when a task must be delayed for a certain number of system ticks or is to wait for an event to occur with a timeout. In this case, this field contains the number of system ticks that the task is allowed to wait for the event to occur. When this value is zero, the task is not delayed.

OSTCBNext and OSTCBPrev are used to doubly link OSTCBs when tasks are created. This chain of OSTCBs is used by OSTimeTick() (described later) to update the OSTCBDly field for each task.

When uCOS is in the process of updating critical data structures, it must ensure that no interruptions will occur. In this case, interrupts are disabled prior to executing this code and reenabled when it is finished executing. The macros OS_ENTER_CRITICAL() and OS_EXIT _CRITICAL() (in UCOS186C.H) are used to disable and enable interrupts, respectively. These macros are included for portability.

TASK STATES

Figure 1 shows the state-transition diagram for tasks under uCOS. At any given time, a task can be in any one of five states. The nOR-MINI state for a task corresponds to a task that resides in EPROM but has not been made available to uCOS. A task is made available to uCOS by calling OS-TaskCreate(), which can be found in UCOS 18 6C.C. When a task is created, it is made ready to run. If the created task is the highest priority task that is ready to run, it is immediately given the RUNNING state, and the task's code is executed by the CPU. While the task is running, it may request from uCOS to wait for an event to occur (PEND) or delay execution for a number of system ticks.

When a task requests such a service from uCOS, the next highest priority task is given control of the CPU. When the event PENDed on occurs, the task is again made ready to run and if it is the highest priority task, it is again given control of the CPU. Other services are available to user tasks to notify other tasks that certain events have occured. A running task may be interrupted, in which case the interrupt-service routine (ISR) takes control of the CPU. The ISR may make one or more tasks ready to run by signaling that one or more events occurred. In this case, before returning from the ISR, uCOS determines if the interrupted task is still the highest priority task that is ready to run. If a higher priority task is made ready to run by the ISR, the previous task is preempted and the new highest priority task is resumed. Otherwise, the interrupted task is resumed.

When all tasks are either waiting for events or delayed for a number of system ticks, uCOS executes an idle task, OSTaskIdle().

TASK CREATION

As previously mentioned, tasks are created by calling OSTask-Create(). Tasks can be created prior to the start of multitasking or by a running task. A task cannot be created by an ISR. OSTaskCreate() is passed four arguments, task is a pointer to the task's code. data is a pointer to a user-definable data area, which is used to pass arguments to the task. pstk is a pointer to the task's stack area, which is used to store local variables and CPU registers during an interrupt. The size of this stack is defined by the task requirements and the anticipated interrupt nesting. Determining the size of the stack involves knowing how many bytes are required to store the local variables for the task, all nested functions, and the requirements for interrupts (accounting for nesting). p is the task priority. A unique priority number must be assigned to each task--the lower the number, the higher the priority.

When a task is created, it is assigned an OS_TCB, which is then initialized. The task stack is initialized to make it look as if an interrupt occurred and the CPU registers were saved onto it. The stack frame of a created task is shown in **Figure 2**. A pointer to the task's OS_TCB is placed in OSTCBPrioTbl[] (UCOS.C) using the task priority as the index. The task's OS_TCB is then inserted in a doubly linked list with OSTCBList (UCOS.C) pointing to the most recently created task. The task is then inserted in a ready list and the scheduler is called to determine if the created task is now the highest priority task that is ready to run.

TASK SCHEDULING

Task scheduling is performed by OSSched() (UCOS.C). uCOS's task-scheduling time is constant regardless of the number of tasks created in an application (up to 63 user tasks). Each task is assigned a unique priority level between zero and 63. Task-priority 63 is always assigned to uCOS's idle task when the kernel is initialized.

Task priorities are grouped (eight tasks per group) in OSRdyGrp (UCOS.C). Each bit in OSBdyGrp is used to indicate that a task in a group is ready to run. When a task is ready to run, it also sets its corresponding bit in the ready list table, OSRdyTbl[]. To determine which priority (and thus which task) will run next, the scheduler determines the lowest priority number that has its bit set in OSRdyTbl[ ]. Tasks are made ready to run by executing the following section of code:

```
OSRdyGrp |=OSMapTbl[p >> 3];
OSRdyTbl[p >> 3] |=OSMapTbl[p & 0x07];
```

This code sets the ready bit of the task in OSRdyTbl[8] based on its priority number, p. Similarly, a task is removed from the ready list by executing the following section of code:

```
if ((OSRdyTbl [p >> 3] &=
 ~OSMapTbl[p & 0x07] == 0)
 OSRdyGrp &= ~OSMapTbl[p >> 3];
```

and clears the bit in OSRdyGrp if all tasks in a group are not ready to run. Instead of scanning through the table starting with OSRdyTbl[O] to find the highest priority task that is ready to run, a table-lookup method is used. OSUnMapTbl[256] is a priority resolution table. Eight bitsare used to represent tasks that are ready. The least significant bit has the highest priority. Using this byte to index the table returns the bit position of the highest priority bit set, a number between zero and seven. To avoid a context switch, OSSched() verities that the highest priority task is not the current task.

All of the code in OSSched() is considered a critical section. Interrupts are disabled to prevent ISRs from setting the ready bit of one or more tasks while the highest priority task that is ready to run is being determined. OSSched() could have been written in assembly language, since task-scheduling time and interrupt latency are directly proportional to the execution time of this function. This code was kept in C to make it more readable for the purpose of this article.

CONTEXT SWITCHING

When a context switch from OSSched() is required, OS_TASK_SW() is executed. OS_TASK_SW(), which is actually a macro defining an INT instruction, vectors to the assembly language ISR OSCtxSw (UCOS186A.ASM), which performs the context switch. Remember that a task stack is initially set up to look as if an interrupt occurred (see **Figure 2**). OsCtxSw expects the current task's stacks and the new highest priority task that is ready to run to look pretty much the same. The INT instruction pushed the processor's PSW and the current task's code segment and offset onto the stack. OSCtxSw starts by saving the rest of the current task's context onto the stack. The processor's stack pointer for the current task is finally saved in the current task's TCB. The stack pointer of the next task to run is restored from its TCB (the new task's context is restored). At this point, the lower priority task's information is stored on the stack. OSCtxSw finally executes a return from interrupt to resume execution of the highest priority task.

INTERRUPT PROCESSING

When an interrupt occurs, the processor will automatically save the current PSW, CS, and IP onto the current task's stack, then fetch the interrupt vector from the interrupt-vector table. If an ISR needs to use the kernel's services, it should immediately enable interrupts to allow interrupt nesting (and reduce interrupt latency) and call OSIntEnter() (UCOS.C) to notify uCOS that it has started servicing an interrupt. The user's ISR code should follow, followed by a call to OSIntExit() (UCOS.C). The ISR code must be written in assembly language under uCOS as follows:

```
ISRx PROC FAR
STI                    ; Enable interrupts
PUSHA                  ; Save CPU's context
PUSH ES
CALL _OSIntEnter       ; Notify uCOS of ISR
CALL _User ISRCode     ; Execute user code
CALL _OSIntExit        ; Notify uCOS of
                       ;end of ISR
POP ES                 ; Restore CPU's
                       ; context
POPA
IRET                   ; Return from
                       ; interrupt
```

OSIntEnter() is used to notify uCOS that an ISR has started, allowing uCOS to keep track of the interrupt-nesting level. uCOS allows nesting of up to 255 interrupts. When the ISR completes, it calls OSIntExit(), which decrements the nesting level. When all interrupts have completed, uCOS finds the highest priority task that is ready to run and switches to that task. Notice that OSIntExit() calls OSIntCtxSw() instead of OSCtxSw. This call is made because uCOS allocates local storage for variables on the stack that are used by OSinExit(). These variables must be deallocated prior to returning to the interrupted task or the highest priority task.

SYSTEM TICK

UCOS allows tasks to suspend execution for a number of system ticks or wait until events occur with a timeout. A system tick is typically provided by a periodic interrupt. The time between interrupts is application specific and is typically between 10 milliseconds and 100 milliseconds--the faster the tick rate, the higher the overhead imposed on the system. Overhead imposed by uCOS is about 3% (worst case) when the tick rate is 20Hz (50 milliseconds). The ISR OSTickISR (UCOS186A.ASM) shows how to implement a system-tick interrupt under uCOS. Notice that OSTime-Tick() is called. This function is used to decrement the OSTCBDly field for each OS_TCB if it is nonzero. When OSTBDly is decremented to zero, the task is made ready to run. The execution time of OSTimeTick() is directly proportional to the number of tasks created in an application (approximately 15,000 bus cycles worst-case with 63 tasks). Before returning to the interrupted task (or the highest priority task that is ready to run) an INT OF2H is performed to chain into the IBM-PC's BIOS-tick ISR (used only for the example). When the BIOS returns from this interrupt, OSTickISR properly terminates the ISR as required by uCOS.

SYSTEM SERVICES

The uCOS kernel offers a number of services to the programmer. In next month's continuation, we'll look at most of the uCOS services. It should be mentioned for the

impatient, however, that one of the most basic kernel services is to allow a task to delay execution for a number of system ticks. The function that performs this operation is called OSTimeDly(), which is found in UCOS.C.

The kernel is initialized by calling OsInit(), also in UCOS.C, which is also where the idle task is created. You must specify a pointer to the idle stack and the maximum number of tasks in your application. After calling OSInit(), you may create your application tasks. Multitasking starts when you call OSStart(). OsInit() must be called before OSStart() and you must create at least one of your tasks prior to calling OSStart(). OSStart() will execute the highest priority task that you created.

PERFORMANCE ISSUES

A number of criteria can alter the performance of a real-time kernel. Context-switch time, interrupt latency, system-services execution time, and task-scheduling time all have some effect.

The context-switch time is really determined by the microprocessor and typically doesn't change from one kernel to another. Saving and restoring a processor's context depends on how many registers it has. In my experience, it's not very meaningful when a kernel vendor quotes performance based on the context-switch time.

An important criterion for a multi-tasking kernel is how long it takes to respond to an interrupt and start executing user code for the ISR. This procedure is known as interrupt latency. A real-time kernel always disables interrupts when manipulating critical sections of code. For uCOS, interrupts are disabled for a maximum of 450 CPU bus cycles. In addition to the period when interrupts are disabled, if uCOS services are required, you must also include the overhead for an ISR preamble. uCOS requires an additional 200 cycles before it starts executing user code. Interrupt latency for uCOS is thus about 650 bus cycles, or 65 microseconds for a 10MHz 80188 system.

Another important performance issue in a real-time kernel is determinism. Kernel services should be deterministic by specifying how long each service call will take to execute. The execution time of uCOS's kernel services does not depend on the number of tasks in an application. This characteristic is desirable, since you don't want the kernel's performance to degrade as the number of tasks in an application increases. **Table 1** lists the execution time of the services described in this article. It was obtained by inspecting the code generated by the compiler and adding up the number of cycles for each instruction. An Intel 80188 processor with zero wait states is assumed. To find the execution time of each kernel service, simply divide the number of cycles given by the processor bus frequency.

The time required by uCOS to find the highest priority task that is ready to run is constant. If the current task has the highest priority and is ready to run (for example, no context switch is required) then OSSched() executes in about 250 bus cycles. If a context switch is required, execution time is about 500 bus cycles.

MEMORY REQUIREMENTS

Given the limited amount of memory typically found in embedded systems (especially with 8-bit microprocessors) the amount Of program and data memory a kernel requires must be kept to a minimum. The RAM requirements for a small model 80188 implementation of uCOS is fairly simple to calculate:

RAM = uCOS requirements +
Task stacks +
Storage of OS_TCBs for all tasks
= 150 + SUM(Task Stacks) +
(OS_TCB size) * (N + 1)

where N is the number of tasks created for the application. uCOS requires 150 bytes for its internal data structures and variables. The size of an OS_TCB is 10 bytes and one extra OS_TCB is required to hold the idle task. The total stack requirement is the sum of the stack requirement for each task. For example, a 20-task application using 256 bytes for each stack would require:

RAM = 150 + 10*(20 + 1) + 21 *256;
= 5,736 bytes.

uCOS requires less than 1,000 bytes of EPROM, excluding the kernel services that I'll describe next month. Our example application requires an additional 300 bytes (excluding the library functions).

StiLL TO COME

Next month, we'll conclude our look at uCOS with a tour of the system services. If you're planning to use the kernel in a project, you might want to hold off on your customization efforts until you examine those services. If you're in a hurry, the code presented this month allows you to run the example shown in **Listing 1**, TEST1.C. This code creates five tasks. Each task delays for one BIOS tick (approximately 55 milliseconds) and displays its priority number at random positions on the screen. The same code is used for each task and the displayed task priority is passed to the task when it is created through the data argument. This type of code can be used for multiple instances of a task as long as the task is reentrant. The code in the example works--even though the functions used are not reentrant (trust me on this). Be careful when using floating-point math in your application--Turbo C + +'s math library is nonreentrant.

The selection of services and other system parameters are based on my experience and seemed a good fit for embedded microcontroller applications. Your experience may be different. I've provided the source code so you can make your own modifications. For example, all of the projects that I've worked on have had less than 35 tasks. Hence, the

kernel limit of 63 tasks seemed quite adequate. If needed, however, uCOS can easily be upgraded to handle up to 127 user tasks. To make such changes easier, I've stressed readability over "optimization." I hope you find the results useful.

BY JEAN J. LABROSSE

Jean J. Labrosse has a Master's degree in Electrical Engineering from the University of Sherbrooke in Quebec, Canada, and has been developing real-time software for over 10 years. He is intimately familiar with the Z-80, 680x, 80x86, 680x0, 6805 and 68HC11 microprocessors. Labrosse is employed by Dynalco Controls in Fort Lauderdale, Fla., where he designs control software for large industrial reciprocating engines.

BIBLIOGRAPHY

Bal Sathe, Dhananjay. "Fast Algorithm Determines Priority," India EDN, Sept. 1988, p.237.

iAPX 186/188 User's Manual and Programmer's Reference. Santa Clara, Calif.: Intel Corp., 1983.

@@**Listing 1**

A multitasking test program.

```
#include "INCLUDES.H"

#define          OS_MAX_TASKS      10

#define          STK_SIZE          500

OS_TCB           OSTCBTbl[OS_MAX_TASKS];

UWORD            OSIdleTaskStk[STK_SIZE];

UWORD            Stk1[STK_SIZE];

UWORD            Stk2[STK_SIZE];

UWORD            Stk3[STK_SIZE];

UWORD            Stk4[STK_SIZE];

UWORD            Stk5[STK_SIZE];

char             Data1 = '1';
```

```
char            Data2 = '2';
char            Data3 = '3';
char            Data4 = '4';
char            Data5 = '5';
void far         Task(void *data);
void interrupt (*OldTickISR)(void);
void main(void)
{
UBYTE err;
clrscr();
OldTickISR = getvect(0x08);
setvect(UCOS, (void interrupt
 (*)(void))OSCtxSw);
setvect(OxF2, OldTickISR);
OSInit(&OSIdleTaskStk[STK_SIZE],
 OS_MAX_TASKS);
OSTaskCreate(Task, (void *)&Data1,
 (void *)&Stk1[STK_SIZE], 1);
OSTaskCreate(Task, (void *)&Data2,
 (void *)&Stk2[STK_SIZE], 2);
OSTaskCreate(Task, (void *)&Data3,
 (void *)&Stk3[STK_SIZE], 3);
```

```
OSTaskCreate(Task, (void *)&Data4,
 (void *)&Stk4[STK_SIZE], 4);
OSTaskCreate(Task, (void *)&Data5,
 (void *)&Stk5[STK_SIZE], 5);
OSStart();
}
void far Task(void *data)
{
 setvect(0x08, (void interrupt
  (*)(void))OSTickISR);
 while (1) {
     OSTimeDly(1);
     gotoxy(rand() % 79 + 1, rand() % 25 + 1);
     putch(*(char *)data);
     if (kbhit()) {
         setvect(0x08, OldTickISR);
         exit(0);
     }
  }
}
```